Technical Document 2282
April 1992

An Implementation of the MVDR Beamformer on the Intel iWarp System

J. Z. Lou

NRaD

Technical Document 2282
April 1992

An Implementation of the MVDR Beamformer on the Intel iWarp System

J. Z. Lou

NAVAL COMMAND, CONTROL AND OCEAN SURVEILLANCE CENTER
RDT&E DIVISION
San Diego, California 92152–5000

J. D. FONTANA, CAPT, USN
Commanding Officer

R. T. SHEARER
Executive Director

ADMINISTRATIVE INFORMATION

The work described in this report was sponsored by the Office of Naval Technology, Office of the Chief of Naval Research. The work was done under program element 0602314N, accession number DN308291.

Released by
G. L. Mohnkern, Technical Staff
Signal and Information
Processing Division

Under authority of
J. A. Roese, Head
Signal and Information
Processing Division

ACKNOWLEDGMENTS

The author thanks Dr. Gary Mohnkern for his suggestions and encouragement on this work. The author also thanks Dr. Aram Kevorkian and Tom Adams for some useful discussions.

LH

CONTENTS

TABLES

1 INTRODUCTION

Beamforming is a classical technique used in array signal processing to determine the location of a source that is radiating energy. A large number of sensors needs to be used to obtain sufficient gain and thus accurately detect a distant target. The computational complexity of a beamformer increases as the number of sensors used increases. Parallel processing is the obvious choice when a large number of sensors is used. In section 2, we give a brief description of beamforming and the Minimum Variance Distortionless Response (MVDR) beamformer. In section 3, we discuss the iWarp architecture and the node connection topologies we used. Sections 4, 5, and 6 discuss implementations of Cholesky, QR, and MVDR algorithms, respectively. Section 7 presents performance measurements from those implementations. The last section gives a brief summary and discusses some future investigations. The complete MVDR code for running on a fully connected network of four processors is also included as appendix A.

2 BEAMFORMING AND THE MVDR BEAMFORMER

In this section, we give a brief description of beamforming and a derivation of the MVDR beamformer. For a detailed discussion on the subject, see references 1 and 2. We assume a plane-wave signal $s(t, x)$ is propagating in a medium with the form at a spatial location z and time t:

$$x(t) = s\left(t + \frac{z \cdot \xi_0}{c}\right),$$

where $-\xi_0$ is the direction of the propagating wa and c is its speed. We also assume there is an array of n sensors present in the medium. Each sensor will record the acoustic field at its spatial position with little interference with each other. Thus the waveform measured at the spatial position z_i of the ith sensor, denoted by $x_i(t)$, is given by

$$x_i(t) = s\left(t + \frac{z_i \cdot \xi_0}{c}\right) + N_i(t),$$

where $N_i(t)$ is additive noise measured at z_i.

In using beamforming to determine the direction ξ_0 of an acoustic source, the outputs of the sensors are summed with weights and delays to form a beam $y(t)$:

$$y(t) = \sum_i a_i x_i(t - \tau_i).$$

The basic idea of beamforming is to adjust the sensor delays so that a signal presumed to be propagating in direction $-\xi_0$ will be reinforced, and signals propagating from other directions will not. The ideal case is that each sensor delay τ_i cancels the signal delay $z_i \cdot \xi_0/c$ so the signal would be completely reinforced. To search for the source signal, the energy in the beam $y(t)$ is computed from many directions-of-look k by manipulating the delays τ_m. The maxima of this energy as a function of k correspond to the acoustic sources.

To compute the beam energy in the frequency domain, we take the Fourier transform of the beam. For a single propagating signal, the Fourier transform of the beam, denoted $Y(f, k)$, is given by

$$Y(f, k) = S(f) \sum_i a_i exp[-j2\pi(f/c)z_i \cdot (k - \xi_0)],$$

where $S(f)$ is the Fourier transform of the signal $s(t)$.

The energy in the beam can be computed by evaluating $\int |Y(f, k)|^2 df$. For a narrowband signal with all its energy concentrated at the frequency f_0, the beam energy $P(k)$ is given by

$$P(k) = |Y(f_0, k)|^2 = \sigma_s^2 \left| \sum_i a \cdot exp\left[-j\frac{2\pi f_0}{c} z_i \cdot (k - \xi_0)\right]\right|^2,$$

where σ_s is a normalizing factor of the signal.

It is now convenient to write equation 1 in matrix form. Define the column vector X to be the temporal Fourier transforms of the array outputs (the signal data vector) and the "steering vector" A to have elements

$$A_i = a_i^* exp\left[j\frac{2\pi f}{c} z_i \cdot k\right].$$

Then $Y(f, k) = A'X$, where $'$ denotes the conjugate transpose. The energy in the beam when steered in direction k is given by

$$P(k) = E[|Y(f,k)|^2] = E[|A'X|^2] = A'R_c A\,, \tag{1}$$

where $R_c = E[XX']$ is the spatial correlation matrix of the sensor outputs. For a general narrowband signal, its spatial correlation matrix R_c in the presence of noise can be written as (see reference 1)

$$R_c = E[(X + N)(X + N)']\,,$$

where X and N are vectors related to array outputs of signal and noise, respectively. Clearly, the quantity $E(XX')$ is hard to compute. In practice, R_c is often estimated by XX', where X is an $m \times n$ matrix with $m \leq n$. The row dimension m is the number of sensors and the column dimension n is the number of "snapshots" in time.

MVDR beamforming is a high-resolution array signal processing algorithm. It is derived by finding the steering vector A that yields the minimum beam energy subject to the constraint that $A'E = 1$, where E represents an ideal plane wave corresponding to the direction-of-look. The purpose of the constraint is to fix the processing gain in the direction-of-look to be unity. The minimization of the resulting energy thus reduces the contributions to this energy from sources and/or noises not propagating in the direction-of-look. This constraint minimization problem can be converted to a global minimization problem by using a Lagrange multiplier. Thus we need to find a vector A that minimizes the quantity

$$A'R_c A + \alpha(A'E - 1)\,.$$

It is not hard to find the solution A to be

$$A = \frac{R_c^{-1}E}{E'R_c^{-1}E}\,.$$

Therefore, the energy in the beam when steered in the direction-of-look determined by E becomes (see equation 1)

$$P_{MVDR} = \frac{1}{(E'R_c^{-1}E)}\,. \tag{2}$$

Equation 2 provides the quantity that we need to compute on a distributed-memory machine. To find a radiating source, we compute the spatial energy spectrum by evaluating equation 2 for many direction-of-look Es and determine the location of local maxima.

In general, the correlation matrix R_c is complex. For the simplicity of data structures, but without loss of generality for our purpose, we used real matrices in our parallel algorithm implementations. Clearly, we also need to assume that the matrix R_c is invertible. In our implementation, we assume X is full-rank. One approach to compute the right-hand side of equation 2 is to do a Cholesky factorization on the matrix R_c. Another approach is not to form the matrix $R_c = XX'$ and compute R_c^{-1}. Instead, a QR factorization on X is performed. The second approach has some advantage in terms of numerical stability. We implemented both parallel Cholesky and parallel QR algorithms and compared their performances. For the MVDR beamforming computation, we used the QR approach.

3 THE iWARP ARCHITECTURE AND INTERPROCESSOR COMMUNICATIONS

The iWarp system is a distributed-memory, scalable, multiple-instruction, multiple-data (MIMD) parallel computer. The system is equipped with an interesting communications library that supports systolic communications between its processors. Since the systolic communication mechanism on iWarp offers "door-to-door" data transfer between CPUs on different processors instead of going through local memories, the iWarp system should be ideal for fine-grain (communications-intensive) applications, for example the digital signal and image processing applications. The system we used has 64 processors (or nodes) that are physically connected into a two-dimensional toroidal mesh. The front-end machine is a Sparc workstation that contains a Sun Interface Board (SIB). The SIB can be used either as a special I/O node or just as other nodes on the system.

Each iWarp node consists of three distinct components: a computation agent, a communication agent, and a local memory unit. The computation agent performs the normal programmed computation. The communication agent handles the I/O from/to adjacent nodes. The local memory unit provides access to local memory for both the computation and communication agents. Each node on the system has a 0.5-megabyte base memory, which can be upgraded in 0.5-megabyte increments. Independent communication and computation agents make it possible to overlap communication and computation. Nonadjacent nodes in the array can communicate *without* necessarily disturbing the computation on intermediate nodes.

Each iWarp processor has an upper-limit speed (or "peak performance") of 20 MFLOPS on single precision floating-point operations and 10 MFLOPS on double precision operations. The bandwidth of interprocessor I/O is 40 megabytes per second. So roughly speaking, a minimum of four single precision operations are needed on each word (4 bytes) of data being sent to gain through parallelism.

Interprocessor communications are realized using the message-passing mechanism provided by the PathLib library on the system (references 3 and 4). To do message passing in a C program, one first needs to create connections. The message passing along the created connections is done by calling a send or receive function in the PathLib. Unlike some other parallel systems (e.g., the Intel Touchstone), the task of routing a certain user-specified connection topology on the iWarp is left entirely to the programmer. In other words, the programmer is responsible for explicitly specifying the routes to be established for the connection topology; the programmer must also appropriately allocate resources required for the type of connection he chooses. This lack of "automatic routing by the system" clearly puts some lower level programming burden on the programmer. The gain from this kind of system design is the flexibility and efficiency to be explored by a programmer for his particular applications. Though there are some programming tools available for some simple image processing applications, it is difficult, at least at this time, to get decent performance on the iWarp for most signal and image processing algorithms by using these tools. We did all our implementations in C using the PathLib.

To establish a communication channel between a pair of iWarp nodes, called the logical channel in the iWarp system manual, a few things must be done: initializing the PathLib, allocating Pathway Control Table (PCT), requesting/accepting connections, and identifying incoming connections. Some care is needed to avoid PCT allocation conflict when one wants to set up a richly connected topology.

To send or receive a message over a logical channel, the programmer needs to bind an input gate or output gate to a proper handler, which is returned by the request or accept connection function calls. A message can be delimited by a header and a trailer. Though it may not be necessary to use a header and a trailer for a simple communication style (e.g., a unidirectional ring connection), they are useful for more complex communication requirements.

In our implementations, we used a unidirectional ring topology and a fully connected network topology. In the unidirectional ring topology, each processor has a unidirectional communication channel with its two neighbors. Each processor can send messages to its downstream neighbor and receive messages from its upstream neighbor once the direction of the ring is determined. This type of sparsely connected topology is suitable for applications that require pipelining communications. The communication overhead will be large if a processor wants to send messages to a distant processor on the ring. In the fully connected topology, we set up a bidirectional connection between each pair of processors. Thus only nearest neighbor communications are required with such a topology. There are two reasons for choosing these two extreme cases of connection topology: (1) we want to test the feasibility of setting up various connection topologies; the success of making these two connections should give us the experience for making a variety of other interesting connection topologies, and (2) we are also interested to see the performance difference in our applications using these two connection topologies.

4 A PARALLEL CHOLESKY ALGORITHM IMPLEMENTATION

As mentioned in section 2, the Cholesky factorization can be used to compute the right-hand side of equation 2. Since the correlation matrix R_c is symmetric (we only consider the real matrix), it has the Cholesky factorization $R_c = GG'$, where G is a lower triangular real matrix.

To design a parallel algorithm for computing the Cholesky factorization on an unidirectional ring topology, we first need to make a decision on how to partition and distribute the matrix. We choose to partition the matrix by columns. Let us assume we want to use p processors on the system. A partition of an $m \times n$ matrix (for the simplicity of notation, we assume n can be divided by p) could be

$$R_c = [R_1, R_2, \cdots, R_k] ,$$

where R_is are $m \times p$ submatrices and $kp = n$. We use the letter k in all our algorithms to be the number of column partitions. We distribute the jth columns ($j = 1 \cdots p$) of the submatrices R_i ($i = 1, \cdots k$) onto the processor j. Thus the processor j will hold columns with indices j, $j + 1$, $\cdots$, $j + (k - 1)p$. A more concrete example should explain this approach better. Suppose we have a matrix written in column notation

$$A = [a_1, a_2, \cdots, a_8] ,$$

where a_is are columns of A. After partitioning and distributing the matrix A onto four processors using our approach, processor 1 contains a_1 and a_5, processor 2 contains a_2 and a_6, processor 3 contains a_3 and a_7, and processor 4 contains a_4 and a_8.

This approach of partition and distribution of matrix columns is clearly suitable for a column-oriented parallel Cholesky algorithm, where $m = n$.

Comparing the jth columns of both sides of the equation $R_c = GG'$, we get the vector equation

$$R_c(:, j) = \sum_{i=1}^{j} G(:, i)G(j, i) ,$$

where : is a range symbol that, when no range is explicitly specified, implies the entire row or column dimension. We obtain from the above equation

$$G(j : n, j)G(j, j) = R_c(j : n, j) - \sum_{i=1}^{j-1} G(j : n, i)G(j, i) \equiv a(j : n) .$$

Since $G(j, j) = a(j)$, it follows that $G(j : n, j) = a(j : n)/\sqrt{\alpha(j)}$. So a column version of the (nonparallel) Cholesky algorithm is as follows:

Column Choleksy Algorithm

```
for j = 1 : n
    a(j : n) ← R_c(j : n, j)
    for i = 1 : j − 1
        for k = 1 : j − 1
            a(k) = a(k) − G(k, i)G(j, k)
        end
    end
    G(j : n, j) ← a(j : n)/√a(j)
end
```

We now describe a parallel Cholesky algorithm based on the above column algorithm, which can be found in reference 5. Note that in the column algorithm, we need to compute

$$a(j:n) = R_c(j:n,j) - \sum_{i=1}^{j-1} G(j:n,i)G(j,i) ,$$

followed by the scaling $G(j:n, j) \leftarrow \alpha(j:n)/\sqrt{\alpha(1)}$. Suppose each of the p processors holds k column vectors of R_c as described at the beginning of this section. The task of each processor is to overwrite these k columns with the nonzero portion of the corresponding columns of the Cholesky factor G. Notice that a column $G(j : n, j)$ generated by one processor is generally needed by all the other processors. So a column $G(j : n, j)$ generated by a processor $Proc(i)$ will be circulated around the ring. At each stop, the visiting $G(j : n, j)$ is incorporated into all the local $\alpha(i)$ for $i \geq j$.

By counting the number of received $G(j : n, j)$ columns, a processor can determine whether it is its turn to generate a $G(j : n, j)$ column. For example, if $Proc(i)$ has received $i - 1$ columns and $i \in \{i, i + p, \cdots, i + (k - 1)p\}$, then it knows that it is its time to generate $G(i : n, i)$. Here is the ring parallel algorithm for the processor i:

Ring Parallel Cholesky Algorithm

```
s ← 0, j ← 0, j1 ← 1;
last ← i + (k − 1)p;
while j ≠ last
   if s + 1 ∈ {i, i + p, ···, last}
      j ← s + 1;
      Generate G(j : n, j) in g_loc(j : n) and copy it into R_loc(j : n, j1);
      if s < k
         send(g_loc(j : n), right);
         update R_loc(:, j1 + 1 : k);
         s ← s + 1;
      end
      j1 = j1 + 1;
   else
      receive(g_loc(s + 1 : k, left);
      if s + 1 ∈ {right, right + p, ···, right + (k − 1)p}
         send(g_loc(s + 1 : n), right);
      end
      s ← s + 1;
      update R_loc(:, j1 : k);
   end
end
```

A few remarks on the variables appearing in the above algorithm are in order. At the beginning of each *while* loop, the variable s indicates the largest global column index among those columns that have been overwritten by their corresponding G columns. The variable j is used to simplify the notation for $s + 1$. The variable j_1 points to the next

column of R_{loc} that will produce a $G(j:n,j)$ column. Also note that a processor does not send the circulating g_{loc} to its right neighbor if its right neighbor is the generator of g_{loc}.

We implemented the above ring parallel algorithm on the iWarp system with a maximum of 16 nodes. It is possible to write a program that is scalable on the number of processors. The amount of physical memory on each node does put a constraint on the size of the problem we can test, and thus the number of processors to use. The current software on the system supports the message passing in the unit of a word in a C program (see reference 6). We therefore wrote a few functions that enable us to send and receive a message with a length of arbitrary number of bytes. After a ring connection of nodes are set up using logical channels, we assign each node a ring ID. The above parallel algorithm can thus be implemented on the system without any modification.

5 A PARALLEL QR ALGORITHM AND ITS IMPLEMENTATION

QR factorization is another way to compute the quantity in equation 2. Note that the correlation matrix is the product of the data matrix X with its transpose. So we can do the QR on X, which, in our implementation, is assumed to be a full-rank real matrix. As will be shown in the next section, we need to store both Q and R factors to compute the MVDR beamforming. There are several approaches to do the QR factorization on the matrix. Since we assume X is full-rank, we can use a QR algorithm without column-pivoting. We use the Householder transformation method, which has been shown to be numerically stable (reference 7). Given a matrix $A \in \Re^{m \times n}$ with $m \geq n$, the sequential algorithm in reference 7 as follows:

The Householder QR Algorithm

for $1 : n$
  $v(j : m) \leftarrow$ **house**$(A(j : m, j))$;
  $A(j : m, j : n) \leftarrow$ **row.house**$(A(j : m, j : n), v(j : m))$;
  if $j < m$
    $A(j + 1 : m, j) \leftarrow v(j + 1 : m)$;
  end
end

In the algorithm above, the function **house(x)**, where **x** is an n-vector, computes n vector v such that $(I - 2vv'/v'v)x$ is zero in all but the first component; the function **row.house** (B, x) overwrites the matrix B with PB, where $P = I - 2vv'/v'v$.

In our ring QR implementation, the input matrix is partitioned and distributed among processors in the same way as we did for the Cholesky factorization. With some modifications to the ring Cholesky algorithm, we have the following ring QR algorithm for the processor i:

A Ring Parallel QR Algorithm

$s \leftarrow 1, j \leftarrow 1$;
$last \leftarrow i + (k - 1)p$;
$next\_rid \leftarrow (i + 1)\ mod(p)$;
$next\_last \leftarrow next\_rid + (k - 1)p$;
while $s < n$
  if $s \in \{i, i + p, \cdots, last\}$
    $v(j : m) \leftarrow$ **house**$(A_{loc}(j : m, j))$;
    Store $v(j : m)$ into $Q(j : m, j)$;
    $send(v(j : m), right)$;
    $A_{loc}(j : m, j : k))$
     $\leftarrow$ **row.house**$(A_{loc}(j : m, j : k)), v(j : m))$;
    $s \leftarrow s + 1, j \leftarrow j + 1$;

—continued

```
    else
        receive(v(j : m), left);
        if s ∉ {next_rid, next_rid + p, ···, next_last}
            send(v(j : m), right);
        end
        Store v(j : m) into Q(j : m, j);
        if s ≤ last
            A_loc(j : m, j : k)
             ← row.house(A_loc(j : m, j : k)), v(j : m));
        end
        s ← s + 1;
    end
end
```

Again we need to explain the variables in the above algorithm. The variables s, j, k, and *last* have the same meanings as in the ring Cholesky algorithm. The variables *next_rid* and *next_last* are the ring ID and the largest global column index for the processor on the ring that is to the right of ith processor. A_{loc} is to be overwritten by the partial factor R, and Q is used to store the orthogonal vectors v. The ring QR algorithm has a similar structure to the ring Cholesky algorithm. Since we need to store the matrix R and the whole matrix Q (in the form of vectors v's on each processor), the variable s for the *while* loop now runs from 1 up to the column dimension for the input matrix A. If s is not less than *last*, we only store the received vector v into Q and do not update A_{loc}. In generating $v(j : m)$, we normalized $A_{loc}(j : m, j)$ by its $\| \|_\infty$ norm to avoid overflow or underflow.

The above ring QR algorithm is rich in the level-2 operation of matrix-vector multiplication, but not rich in the level-3 operation of matrix-matrix multiplications; the latter operation is more desirable for reducing the memory traffic on many high-performance architectures. Using the idea of block Householder QR factorization (reference 7), we can modify the above ring QR algorithm to get more level-3 operations. The idea stems from the fact that a product of $n \times n$ Householder matrices $Q = Q_1 \cdots Q_r$ can be written in a form called the WY representation

$$Q = I + WY' ,$$

where W and Y are $n \times r$ matrices. More specifically, given an orthogonal matrix $Q = I + WY'$ and a Householder matrix $H = I - 2vv'/v'v$, we have

$$Q_1 = QH = I + W_1 Y'_1 ,$$

where $W_1 = [W, w]$ and $Y = [Y, v]$ with $w = -Qv/v'v$. For a derivative of this, see reference 8. The basic idea of the block Householder QR is to form a product of Householder matrices using the WY representation and then apply this product to the remaining unupdated columns. The block Householder QR algorithm can be found in reference 7 or 8.

To incorporate the block QR into our ring QR algorithm, we may modify the while loop as follows. When **if** $j \in \{i, i + p, \cdots, last\}$ is true, we only update the jth column in the processor i. The W and Y matrices are updated in each *while* loop. At the end of each *while* loop, we check the global counter s to determine if we need to do a block update on

the local columns $[s\ mod(p)+1, \cdots, k]$. The function **row.house** also needs some obvious modification. The modified ring block QR algorithm is as follows:

A Ring Block Parallel QR Algorithm

$s \leftarrow 1, j \leftarrow 1;$
$last \leftarrow i + (k-1)p;$
$W \leftarrow I, Y \leftarrow I;$
$next\_rid \leftarrow (i+1)\ mod(p);$
$next\_last \leftarrow next\_rid + (k-1)p;$
while $s < n$
  if $s \in \{i, i+p, \cdots, last\}$
    $v(j:m) \leftarrow$ **house**$(A_{loc}(j:m, j));$
    Update W and Y using $v(j:m)$;
    Store $v(j:m)$ into $Q(j:m, j)$;
    send$(v(j:m), right)$;
    $A_{loc}(j:m, j))$
     $\leftarrow$ **row.house**$(A_{loc}(j:m, j)), v(j:m));$
    $s \leftarrow s+1, j \leftarrow j+1;$
  else
    receive$(v(j:m), left)$;
    if $s \notin \{next\_rid, next\_rid+p, \cdots, next\_last\}$
      send$(v(j:m), right)$;
    end
    Update W and Y using received $v(j:m)$;
    Store $v(j:m)$ into $Q(j:m, j)$;
  end
  if $((s\ mod(p)) = 0$ **and** $s < last;$
    $A_{loc}(s+1:m, s/p+1:k)$
     $\leftarrow$ **row.house**$(A_{loc}(s+1:m, s/p+1:k)), I + WY');$
  end
  $s \leftarrow s+1;$
end

We implemented the ring QR on the iWarp system. The modified block version has not been implemented at this moment since we are not sure that our current system can do level-3 operations more efficiently. We also implemented QR on a fully connected network of nodes. An easy modification to the ring QR algorithms produces a QR algorithm for the fully connected topology. We will discuss more about full connection in the next section.

6 A PARALLEL MVDR BEAMFORMER IMPLEMENTATION

We first show what needs to be computed for a MVDR beamformer using the QR factorization. As shown in section 2, we want to compute $(ER_c^{-1}\mathrm{E})^{-1}$. Since $R_c = XX'$ and $X = \mathrm{QR}$, we have

$$\begin{aligned}(E'(XX')^{-1}E)^{-1} &= (E'(X')^{-1}X^{-1}E)^{-1} \\ &= (E'(R'Q')^{-1}(QR)^{-1}E)^{-1} \\ &= ((R^{-1}Q'E)'(R^{-1}Q'E))^{-1} = (z'z)^{-1},\end{aligned}$$

where $z = R^{-1}Q'E = R^{-1}QE$ since Q is symmetric. Thus we see the MVDR computation using QR consists of a QR factorization, a matrix multiplication of Q with the direction-of-look vector E, and a solution of an upper linear triangular system $z = R^{-1}y$.

In the last section, we discussed implementation of QR factorization. Now we want to find a parallel algorithm for the solution of a linear triangular system. Reference 9 gives a detailed discussion of several parallel triangular system algorithms, which are based on the following basic sequential algorithms:

Two Triangular System Algorithms

```
for i = 1 : n                    for j = 1 : n
  for j = 1 : i - 1                x_j = b_j / L_jj
    b_i = b_i - x_j L_ij           for i = j + 1 : n
  end                                b_i = b_i - x_j L_ij
x_i = b_i / L_ii                   end
end                              end
```

The algorithms presented above are for lower triangular systems. But an index reversal in these algorithms can make them work for an upper triangular system. Using the terminology in reference 9, the algorithm on the left is called a scalar-product algorithm, and the algorithm on the right is called the vector-sum algorithm. If our sole task were to solve a triangular system on a parallel machine, we could partition the matrix by either columns or rows. To explore parallelism from the above algorithms, we have the choice of partitioning the work in the inner loop or the outer loop. So there are many possible parallel algorithms. Since we have to solve a triangular system following the parallel QR factorization, we do not have the freedom of choosing a matrix partition. In fact, we must use the column partition of the upper triangular matrix R, which is already distributed among processors after the QR factorization is performed. We also decide to parallelize the inner loop for a relatively simple implementation. It turns out that the algorithm on the above left can be used as the basis to construct an algorithm, called the "fan-in scalar-product" algorithm in reference 9, which parallelizes the inner loop and requires a column partition of the matrix. The resulting algorithm, with "fan-in scalar-product" implemented on a unidirectional ring, is as follows:

The Ring "Fan-In" Triangular System Algorithm

```
for i = 1 : n
    s = 0
    for j = 1 : i - 1
        if j ∈ {rid, rid + p, ···, rid + (k - 1)p}
            s = s + L_ij x_j;
    end
    if i ≠ 1
        new_id = rid - map(i);
        if new_id < 0
            new_id = new_id + p;
        if new_id = 1
            send(s, right);
        if new_id = 0
            receive(buffer, left);
            s = s + buffer;
        end
        if new_id ≠ 0 and new_id ≠ 1
            receive(buffer, left);
            s = s + buffer;
            send(s, right);
        end
    end
    if j ∈ {rid, rid + p, ···, rid + (k - 1)p}
        x_i = (b_i - s)/L_ii
end
```

In the above algorithm, *rid* is the ring ID for node *i*. The *map*(*i*) is the ring ID for the node that contains *i*th column of the matrix. The statements in the if $i \neq 1$ section constitute the "fan-in" operation. In each loop, what the "fan-in" operation does is to send all partial updates of b_i to the node with ring ID map(i). With a unidirectional ring connection, we implemented the "fan-in" by computing a *new_id* that is the "distance" from the current node to the node *map*(*i*). Once this *new_id* is computed, it is easy to decide what each node needs to do in this "fan-in" operation. As can be seen from thi lgorithm, the parallelism comes from computing the partial updates of b_i; the unknown x_is are still computed sequentially.

The main disadvantage of a ring connection of nodes is the communication overhead. For a unidirectional ring with n nodes, its diameter is n 1. For a full connection of n nodes, its diameter is 1. So for a "fan-in" operation, the unidirectional ring connection is less efficient than for a full connection because the former requires more intermediate steps for passing a message around. Though it requires more work to set up a full connection of nodes, at least on the iWarp system, it is easy to modify our QR and triangular system ring program so that they run on a full connection of nodes. We do not list the corresponding algorithms for a full connection of nodes (they are simpler and have similar structures to the ring algorithms). We just give their performance results in the next section.

With the parallel QR and the parallel triangular system subroutines, we can now give the whole program for computing the MVDR beamforming with n direction-of-look vectors $\{E_1, E_2, \cdots, E_n\}$ as follows:

A Parallel MVDR Beamformer Program

Set up connection network of nodes on the system;

Dynamic memory allocations on each node;

Input array output matrix X to node 0;

Distribute appropriate portions of X to each node on the network;

Perform a parallel QR factorization on X;

for $i = 1 : n$

Input direction-of-look E_i to node 0;

Broadcast E_i to each node on the network;

Compute in parallel $z = QE_i$;

Compute in parallel $s = R^{-1}z$ and s^2;

Gather components of s^2 from each node on the network to node 0;

Output $1/s^2$ from node 0

end

We used dynamic memory allocation in the above program. Allocating memory at run time makes the program more robust and storage structures more flexible. We assume that node 0 is the only processor that does external I/O. We also see that no explicit global synchronization is needed to ensure the parallel program function correctly as long as every interprocessor message passing proceeds correctly.

7 PERFORMANCE MEASUREMENTS

In this section, we present performance measurements on the iWarp system of the parallel algorithms we have discussed. All floating-point operations were performed using single precision. Because each node on our current iWarp system has 0.5 megabyte of physical memory, we choose the data matrix X for performance measurements to be 128 $\times$ 128. We first want to point out that, with the current software release (e.g., the compiler) on the system, the iWarp node performance on a C code is low compared with its theoretical (peak) performance. For example, we have run a matrix multiply C program on a single node with available optimizations and obtained a performance of about 0.65 MFLOPS. Hence, we do not expect good MFLOPS performance from our parallel program on the iWarp system. Relatively speaking, the iWarp system has good bandwidth for interprocessor communications. It is interesting to see the speed-ups we can get from our parallel algorithm implementations on the system. For comparison, we also measured corresponding performances on a Convex C-240; we just used one vector processor on that system. Using the usual terminology in parallel processing, speed-up is the ratio of performance on a single processor to the performance on multiple processors, and efficiency is the ratio of speed-up to the number of processors used.

Table 1 lists the performance measurements for a Cholesky factorization using a unidirectional ring connection. It shows that we got a speed-up of 3.3 and an efficiency of 83% using 4 nodes; we got a speed-up of 7.1 and an efficiency of 44% using 16 nodes.

Table 1. Performance measurements for a Cholesky factorization.

	Cholesky Factorization (128 x 128)		
Machine	# Processors	CPU time (seconds)	Speed-ups
iWarp	1	2.00	
iWarp	4	0.60	3.3
iWarp	16	0.28	7.1
Convex	1	0.09	

Table 2 lists the performance measurements for a QR factorization using a unidirectional ring connection. It shows a speed-up of 3.6 and an efficiency of 90% using 4 nodes; a speed-up of 10.3 and an efficiency of 64% using 16 nodes.

Table 2. Performance measurements for a QR factorization.

	Cholesky Factorization (128 x 128)		
Machine	# Processors	CPU time (seconds)	Speed-ups
iWarp	1	7.20	
iWarp	4	2.00	3.6
iWarp	16	0.70	10.3
Convex	2	0.22	

Table 3 lists the performance measurements of a MVDR with QR using a unidirectional ring connection. In performance measurements for MVDR, we use only one direction-of-look vector E in the program. It shows we got a speed-up of 3.3 and an efficiency of 83% using 4 nodes; a speed-up of 5.8 and an efficiency of 34% using 16 nodes.

Table 3. Performance measurements for a MVDR with QR.

Cholesky Factorization (128 × 128)			
Machine	# Processors	CPU time (seconds)	Speed-ups
iWarp	1	11.5	
iWarp	4	3.50	3.3
iWarp	16	2.00	5.8
Convex	1	0.40	

Table 4 lists the performance comparisons for Cholesky, QR, and MVDR using a full connection of four nodes with a unidirectional ring connection. For the Cholesky algorithm, its performance on a full connection of nodes is about 30% faster than its performance on a ring connection. For the QR algorithm, we do not see much performance difference between a full connection and a ring connection. The performance difference for MVDR is about 20% with a full connection over a ring connection.

Table 4. Performance comparisons for Cholesky, QR, and MVDR.

Comparisons between Full and Ring Connection Using Four iWarp Processors		
Application	Type of Connection	CPU time (seconds)
Cholesky	Ring	0.60
Cholesky	Full	0.43
QR	Ring	2.00
QR	Full	1.93
MVDR	Ring	3.50
MVDR	Full	2.80

8 REMARKS AND FUTURE INVESTIGATIONS

The design of iWarp node architecture and the system physical network as a two-dimensional mesh is intended to make it suitable for both fine-grain computations and coarse-grain computations (reference 3). For the matrix size we used, our problem may be seen as a fine-grain application. We think the good speed-ups on Cholesky and QR factorizations using four nodes show the iWarp system's communication bandwidth is fast. It is also reasonable to see that efficiency goes down as the number of nodes increases on a unidirectional ring, since, except for some highly pipelined types of application, the communication overhead increases with a larger ring radius. We expect the efficiency of our application would be greater on a more richly connected topology like a bidirectional ring or a two-dimensional mesh. Due to the tedious work involved in ng a fully connected network of nodes, we only set up a fully connected network containing four nodes to compare its performance with a unidirectional ring. Though we do not see a significant performance difference on the QR factorization, which implies the communication overhead is very small for that application on a four-node ring, we see an obvious performance difference on the MVDR. This difference clearly came from the triangular system solver. It seems to us that the parallel triangular system algorithm is "finer grain" than is the parallel QR algorithm, so the former should perform better on a more richly connected topology.

Compared with the performance of some vector machines in a high-level language code (like C or Fortran), we find our current iWarp system is still not a fast machine. This is not surprising considering the fact that each iWarp node is not a vector processor and the software technology for the system is not very mature yet. At this time, however, we can use the iWarp system to test and evaluate parallel algorithms using different connection topologies and gain parallel programming experience.

We think a few things are worth trying on the iWarp system in the near future. It would be interesting to try a larger size problem and thus use more processors for performance testing, which may require us to upgrade memory on at least some of the processors. We are also interested in developing and implementing systolic algorithms that use two-dimensional mesh—a natural choice for the iWarp system. We would also like to modify and run our application codes on the Intel Touchstone system so that a performance comparison can be made for these two parallel MIMD machines.

9 REFERENCES

1. Johnson, D., "The Application of Spectral Estimation Methods to Bearing Estimation Problems," Proceedings of the IEEE, vol. 70, no. 9, September 1982.
2. Johnson, D. and D. Dudgeon, "Fundamentals of Array Signal Processing," Lecture Notes, Rice University, 1990.
3. Intel Corporation, *iWarp Programmer's Guide*, September 1991.
4. Greer, B., *A Tutorial on Using iWarp PathLib*, Intel Corporation, October 1991.
5. Van Loan, C., "A Survey of Matrix Computation," Theory Center Technical Report, Advanced Computing Research Institute, Cornell University, Ithaca, NY.
6. Intel Corporation, *iWarp C User's Guide*, September 1991.
7. Golub, G. and C. Van Loan, *Matrix Computation*, The Johns Hopkins University Press, 1989.
8. Bischof C. and C. Van Loan, "The WY Representation for Products of Householder Matrices," SIAM J. Scientific and Statistical Computing. vol. 8, no. 1, January 1987.
9. Heath, M. and C. Romine, "Parallel Solution of Triangular Systems on Distributed-Memory Multiprocessors," SIAM J. Scientific and Statistical Computing, vol. 9, no. 3, May 1988.

APPENDIX A

MVDR CODE

```
/*
 * A Head File for MVDR Beamformer
 */

#include <sys/time.h>
#include <stdio.h>
#include <math.h>
#include <iwarp_cc.h>
#include <gates.h>
#include <regnums.h>
#include <asm/gen_asm.h>
#include <asm/pw_asm.h>
#include <pathlib/pl.h>

/* Define the scale of the problem ! */
#define maxcell  4   /* total number of iWarp cells to use */
#define rdim  128  /* row dimension of data matrix A */
#define cdim  128  /* column dimension of data matrix A */
#define srdim  rdim     /* row dimension of storage */
#define scdim  cdim     /* column dimension of storage */


/* Define date type */
typedef float* vector;
typedef vector* matrix;

/* Function declarations */
void main();
void fl_connt();
void de_connt();
void assign_chan();
void send_a_buf();
void receive_a_buf();
void input_A();
void get_space();
void distribute_A();
void distribute_xv();
void qr();
void store_qq();
void gen_g();
int  imax();
void update();
void mvdr();
void tri_solver();
void gather_g();
void print_g();
void print_qq();

/* Other global definitions */
char *malloc();
int _cellid, ringid;
int elsize; /* size of a double precision variable */
int c[maxcell];
```

```
/*
 * A Four-Processor Program Using PathLib Communication to Implement
 * a MVDR beamformer using a Ring Connection.
 *
 * Input:  a data matrix A and a column vector (direction of look) x.
 * Output: x'inv(AA')x ( ' means transpose). x has dimension of row(A).
 *
 * Computation procedure:
 *    1) A = QR, Q = orthogonal, R = upper-triangular;
 *    2) form the quantity  z = inv(R)Q'x;
 *    3) compute  b = z'z;
 *    4) output b.
 */

/* This file contains main(), ring(), de_ring(), send_a_buf(),
   receive_a_buf(). */


#include "mvdr.h"   /* mvdr.h contains 'include files' and
                  global declarations */

long  sec, usec;
int i_chan, o_chan, headers, header_count;
float exe_time;
struct timeval tp1, tp2;
struct timezone tzp1, tzp2;

void main()
{
 /* variables definitions */
  matrix a;  /* data matrix A or data sub_matrix of A */
  matrix qq;  /* triangular Q factor matrix (for storing q1,...,qn) */
  vector xv; /* direction of look column vector */
  int  ncols;
  float  out; /* output value b */

 /* Initialize Pathlib */
  PL_INIT(_cellid, 0xfffc, 0xff);
  pl_rpe_configure(0x0030, 0x00c0, 0x0300, 0x0c00, 0xf000);

 elsize = sizeof(float);
 ncols = cdim/maxcell;

 /* Set up the Ring */
  ring();


 /* Input matrix A and column vector xv to cell 0 and allocate
    space for other cells */
  if (ringid == 0)
    input_A(&a, &qq, &xv, srdim, scdim, ncols); /* input A to cell 0 */
  else
    get_space(&a, &qq, &xv, srdim, ncols); /* allocate space for cells
                                              other than cell 0 */

 /* Distribute columns of A to appropriate cells */
  distribute_A(a, ncols);

 /* Parallel computing QR decomposition begins ... */
  qr(a, qq, ncols);
 /* end of computing QR: R distributed in a's, Q stored in qq  */

 /* Distribute vector xv to all cells */
  distribute_xv(xv, ncols);
```

```
 /* Parallel MVDR begins ... */
  mvdr(a, qq, xv, ncols, &out);

 /* Output MVDR value stored in 'out' */
  if(ringid ==0) {
    out = 1.0/out;
    printf("*** output = %.4f ***\n", ringid, out);
  }

 /* Destroy the ring connection */
  de_ring();

  exit (0);

} /* main */.



/* Set up a ring with four cells */
void
ring()
{
 int next, o_dir;
 int header;
 int row1_last=1, row2_first=8, row=8;


 if (_cellid < row1_last) {
   next = _cellid + 1;
   o_dir = PL_DIR_XR;
   ringid = _cellid;
 }
 if (_cellid == row1_last) {
   next = 9;
   o_dir = PL_DIR_YD;
   ringid = _cellid;
 }
 if (_cellid == row2_first) {
   next = 0;
   o_dir = PL_DIR_YU;
   ringid = 3;
 }
 if (_cellid > row1_last & _cellid != row2_first) {
   next = _cellid - 1;
   o_dir = PL_DIR_XL;
   ringid = 2;
 }


 /* Set up logical channels to form a Ring */
 /* Each cell does an accept and a create connection */

 o_chan = pl_create_connection(GATE0_OUT, o_dir, &next, 1);
 (void) pl_produce_message_header(GATE0_OUT, 0);
 i_chan = pl_accept_connection(GATE1_IN, &header);
 (void) pl_consume_message_header(GATE1_IN, &header);

} /* Ring */


void
de_ring()
{
 /* Terminate the output channel */
  (void) pl_produce_message_trailer(GATE0_OUT);
```

```
  (void) pl_destroy_connection(o_chan, GATE0_OUT);
 /* Terminate the input channel */
  (void) pl_consume_message_trailer(GATE1_IN);
  (void) pl_conclude_connection (i_chan, GATE1_IN);
}


void
receive_a_buf(pbuf, N)
char *pbuf;
int N;
{
 int qt, rd, k, ibuf;
 int *ip = (int *) pbuf;

 qt = N/4, rd = N - qt*4;
 for (k=0; k<qt; k++) {
   ibuf = _receivei(GATE1_IN);
   *ip++ = ibuf;
 }

 if (rd > 0) {
   char *pt;
   char *cp = (char *)ip;
   ibuf = _receivei(GATE1_IN);
   pt = (char *)ibuf;
   for (k=0; k<rd; k++) {
     *cp++ = *pt++;
   }
 }
} /* receive_a_buf */


void
s    a_buf(pbuf, N)
     *pbuf;
int N;
{
 int qt, rd, k, ibuf;
 int *ip = (int *) pbuf;

 qt = N/4, rd = N - qt*4;
 for (k=0; k<qt; k++) {
   ibuf = *ip++;
   _sendi(GATE0_OUT, ibuf);


    (rd > 0) {
   char *pt = (char *)ibuf;
   char *cp = (char *)ip;
   for (k=0; k<rd; k++) {
     *pt++ = *cp++;
   }
   _sendi(GATE0_OUT, ibuf);
 }
} /* send_a_buf */
```

```
/*
 * A few subroutines called by mvdr.c:
 * strcpy(), input_A(), get_space(), distribute_A(), qr(),
 * gather_g(), print_r().
 * a[j][i] is the element on jth column and ith row.
 */

#include "mvdr.h"

void strcpy(s1, s2, size)
char *s1, *s2;
int  size; /* number of chars in the string */
{
 int i;
 for(i=0; i<size; i++)
   s1[i] = s2[i];
}/* strcpy */

void
input_A(pa, qq, xv, rn, cn, cols)
/* rn, cn = row and column dimension of storage */
matrix *pa, *qq;
vector *xv;
int rn, cn, cols;
{
 /* I/O files named 'stdin0' and 'stdout0' must exist in
    the current directory. Using lgo with -s option
    to redirect stdin and stdout to stdin0 and stdout0 */

  matrix mp, mp1;  /* mp, mp1 are to traverse the matrix columns */
  vector  vp;
  int  i, j;

 /* create an array of column pointers storage to store A (and output) */
  *pa = (matrix) malloc(sizeof(vector)*cn), mp = *pa;
  *qq = (matrix) malloc(sizeof(vector)*rn), mp1 = *qq;
  *xv = (vector) malloc(elsize*rdim); vp = *xv;

 /* input matrix A */
  for(j=0; j<cn; j++)
   {
     /* input a column vector */
     mp[j] = (vector) malloc(elsize*rn);
     for(i=0; i<rn; i++)
      {
       /* fscanf(stdin, "%f", &mp[j][i]); */
        if(i==j)
         mp[j][i] = 150;
        else
     mp[j][i] = 1;
      }
   }

 /* input direction of look vector xv */
     for(i=0; i<rn; i++)
       vp[i] = (float) i;

 /* allocate space for Q factor */
 for(j=0; j<rn; j++)
   mp1[j] = (vector) malloc(elsize*(rn - j + 1));
  /* last element of vector qq[j] stores the norm squared of qq_j */

} /* input_A */
```

```
void
get_space(pa, qq, xv, rn, cn)
matrix *pa, *qq;
vector *xv;
int rn, cn;
{
  matrix  mp, mp1;
  int  j;

 /* create an array of column pointers storage */
  *pa = (matrix) malloc(sizeof(vector)*cn), mp = *pa;
  *qq = (matrix) malloc(sizeof(vector)*rn), mp1 = *qq;
  *xv = (vector) malloc(elsize*rn);
  for (j=0; j<cn; j++)
    mp[j] = (vector) malloc(elsize*rn);

 /* allocate space for Q factor */
  for (j=0; j<rn; j++)
    mp1[j] = (vector) malloc(elsize*(rn - j + 1));
    /* last element of vector qq[j] stores the norm squared of qq_j */

} /* get_space */


void
distribute_A(a, cols)
matrix a;
int cols;
{
  matrix  mp = a,
          tmp = a; /* tmp is to arrange columns for cell 0 */
  int  i, j;
  int  num_send = maxcell - ringid - 1,
       num_recv = num_send + 1;
  float  buf[rdim];

  if(ringid==0)
    num_recv = 0;

 /* the matrix A has cdim = maxcell*cols columns */
  for(i=0; i<cols; i++)
   {
     /* send out columns from cell 0 */
     if(ringid==0)
       {
         /* keep 1st one of every maxcell columns */
           strcpy((char *) tmp[i], (char *) mp[i*maxcell], rdim*elsize);
         for(j=0; j<num_send; j++)
           send_a_buf((char *)mp[j+1+i*maxcell], rdim*elsize);
       }

     /* receive and send columns on cells != 0 */
     if(num_recv)
       {
          for(j=0; j<num_recv; j++)
           {
             if(j==0)
               receive_a_buf((char *)mp[i], rdim*elsize);
             else
           {
               receive_a_buf((char *)buf, rdim*elsize);
               send_a_buf((char *)buf, rdim*elsize);
             }
           }
       }
    } /* index i */
```

```
} /* distribute_A */


void
distribute_xv(xv, cols)
vector xv;
int cols;
{
 int num_send = 1, num_recv = 1;
 if(ringid==0)
   num_recv = 0;
 if( ((ringid+1)%maxcell) == 0)
   num_send = 0;
 if(num_recv)
   receive_a_buf(xv, elsize*rdim);
 if(num_send)
   send_a_buf(xv, elsize*rdim);

} /* distribute_xv */



void
qr(a, qq, cols)
matrix a, qq;
int  cols;
{
  /*
   * variable definitions:
   *  last --- global index for the last vector stored in proc(u).
   *  k    --- global index of the column vector currently working on.
   *  j    --- global index of last locally generated g-column.
   *  j1   --- local index of next column of loacal_A that will
   *           produce a g-column.
   */

   int  last = ringid + (cols - 1)*maxcell;
   int  next_rid = (ringid + 1) % maxcell;
                  /* next_rid = ringid for the next cell on the ring */
   int  next_last = next_rid + (cols - 1)*maxcell;
   int  k, j1, kp;
   int  i, nol;
   matrix  mp = a;
   float  *pt, norm;
   float   vp1[rdim]; /* tmp. storage vector */

  /* initializing index and pointer variables */
   k = 0, j1 = 0, kp = 0;
   while (k < rdim - 1)
    {
       kp = (k - ringid) % maxcell;
       if(kp==0) /* its turn comes */
        {
         /* generate a v_column based on kth column and
            copy it to vp1 */
           gen_g(mp[j1], vp1, k);
           store_qq(qq, vp1, k);
           send_a_buf((char *) vp1, (rdim-k)*elsize);
           update(a, vp1, j1, cols, rdim-k);
           k++,  j1++;
        }
       else
         {
           receive_a_buf((char *)vp1, (rdim-k)*elsize);
           kp = (k - next_rid) % maxcell;
           if(kp != 0)
```

```
            send_a_buf((char *) vp1, (rdim-k)*elsize);
          store_qq(qq, vp1, k);
          if(k <= last)
            update(a, vp1, j1, cols, rdim-k);
          k++;
        }
  } /* while */

} /* qr */


void
store_qq(qq, vp1, k)
matrix  qq;
vector  vp1;
int     k;
{
  int  j1, nel;
  float  norm = 0.0;

  nel = rdim - k;
  for(j1=0; j1<(rdim-k); j1++)
    norm += vp1[j1]*vp1[j1];
  qq[k][nel] = norm;
  strcpy((char *) qq[k], (char *) vp1, elsize*nel);

/*  if(ringid==0) {
  printf("k = %d, norm = %.2f\n", k, norm);
  printf("vp1 = \n");
  for(j1=0; j1<nel; j1++)
    printf("%.1f ", vp1[j1]);
  printf("\n");
  printf("qq[0] = \n");
  for(j1=0; j1<nel+1; j1++)
    printf("%.1f ", qq[k][j1]);
  printf("\n");
  }
*/

} /* store_qq */


void
gen_g(vp, vp1, j)
vector vp;
float vp1[];
int  j; /* jth column we are working on */
{
  int  i=0, n=0, ind=0;
  float  max, max2, sign = 1.0, norm = 0.0;

  ind = imax(vp, j, rdim);
  if((max = vp[ind]) == 0)
    printf("max = 0!!!\n");
  max2 = max*max;
  for(i=j; i<rdim; i++)
    norm += vp[i]*vp[i]/max2;
  norm = sqrt(norm);
  if(vp[j] < 0)
    sign = -1.0;
  for(i=j; i<rdim; i++)
   {
     if(i==j)
       vp1[n] = vp[i]/max + sign*norm;
     else
```

```
        vpl[n] = vp[i]/max;
      n++;
    }
} /* gen_g */


/* find the index of max component of v between n1 <= index < n2 */
int imax(v, n1, n2)
float *v;
int n1, n2;
{
 int  i, index = n1;
 float  val = fabs((double) v[n1]);

 for(i=n1; i<n2; i++)
   if(val < fabs((double) v[i]))
    {
      val = fabs((double) v[i]);
      index = i;
    }
 return index;
}/* imax */


void
update(a, vpl, m, cols, nel)
matrix a;
float vpl[];
int  m, cols, nel;
/* m = local column index of first remaining vectors to be updated */
/* nel = number of elements in received vector vpl */
{
 matrix  mp = a;
 int  j=0,
      jcol = rdim - nel; /* column index of received column */
 int  idx, /* global row index of a matrix element */
      jdx, /* global column index of a matrix element */
      jp, ip; /* indices for vector vpl */
 float  norm = 0.0, vx = 0.0;

 for(j=0; j<nel; j++)
   norm += vpl[j]*vpl[j];

 /* update mth to cols-1'th columns: mp[m] -> mp[col-1] */
 for(j=m; j<cols; j++)
  {
    vx = 0.0, ip = 0;
    for(idx=jcol; idx<rdim; idx++)
      {
  vx += mp[j][idx]*vpl[ip];
        ip++;
      }
    vx = 2*vx/norm; /* coeff. */
    ip = 0;
    for(idx=jcol; idx<rdim; idx++)
      {
        mp[j][idx] -= vx*vpl[ip];
        ip++;
      }
  }

} /* update */


void
mvdr(a, qq, xv, cols, out)
```

```
matrix  a, qq;
vector  xv;
int cols;
float  *out;
{
 int  i, j;
 float  cf = 0.0, fbuf = 0.0;
 float  *sol = (float *) malloc(sizeof(float)*cols);
 int  nrecv = 0;

 /* compute Q'xv */
 for(j=0; j<rdim - 1; j++)
  {
    cf = 0.0;
    for(i=0; i<(rdim - j); i++)
      cf += xv[j+i]*qq[j][i];
    cf = 2*cf/qq[j][rdim - j]; /* qq[j][rdim - j] = norm squared
                                  of qq[j] */
    for(i=0; i<(rdim - j); i++)
      xv[j+i] -= cf*qq[j][i];
  }
  /* now xv contains Q'xv */


/* if(ringid == 3)
   for(i=0; i<rdim; i++)
     printf("ringid %d: i = %d, xv[i] = %.2f\n", ringid, i, xv[i]);
*/

 /* compute inv(R)z, z = Q'xv */
 /* each cell contains corresponding components of the right-hand side
    vector computed from Q'xv */

 tri_solver(a, cols, xv, sol);

 *out = 0.0;
 for (i=0; i<cols; i++)
    *out += sol[i]*sol[i];

 printf("ringid = %d, out = %.2f\n", ringid, *out);

 /* collect output to cell 0 */
  fbuf = 0.0;
  if(ringid != 1)
    nrecv = 1;
  if(ringid == 1)
    send_a_buf((char *) out, elsize);
  if(nrecv == 1 && ringid != 0)
   {
     receive_a_buf((char *)&fbuf, elsize);
     fbuf += *out;
     send_a_buf((char *)&fbuf, elsize);
   }
  if(ringid==0)
   {
     receive_a_buf((char *)&fbuf, elsize);
     *out += fbuf;
   }
 if(ringid == 0)
   printf("ringid = %d, total out = %.2f\n", ringid, *out);


} /* mvdr */
```

```
/* for upper-triangular matrix */
void
tri_solver(a, cols, xv, sol)
matrix  a;
int  cols;
vector  xv, sol; /* sol[cols] is the sub-solution vecter */
{
  int  i, j, idx, jdx, nid, mapi, nrecv;
  int  last = ringid + (cols - 1)*maxcell;
  float  tmp = 0.0, fbuf = 0.0;

  for(i=rdim-1; i>=0; i--)
   {
    tmp = 0.0,  fbuf = 0.0;
    mapi = i%maxcell;
    for(j=ringid; j<=last; j+=maxcell) /* for j in mycols */
      if(j>i)
        {
         idx = j/maxcell; /* local col index */
         jdx = i; /* row indox */
         tmp += a[idx][jdx]*sol[idx];
        }

    /* fan-in begins ... */
    if(i != (rdim - 1))
     {
       /* compute new origin */
       nid = ringid - mapi;
       if(nid<0)
         nid += maxcell;
       if(nid == 1)
         send_a_buf((char *)&tmp, sizeof(float));
       if(nid != 1 && nid != 0)
         {
           receive_a_buf((char *)&fbuf, sizeof(float));
           tmp += fbuf;
           send_a_buf((char *)&tmp, sizeof(float));
         }
        if(nid == 0)
         {
           receive_a_buf((char *)&fbuf, sizeof(float));
           tmp += fbuf;
         }
     }
    /* end of fan-in */

    if(!((i-ringid)%maxcell)) /* if i in mycols */
      sol[i/maxcell] = (xv[i] - tmp)/a[i/maxcell][i];
   } /* for i=1 to n */

} /* tri_solver */


void
gather_g(a, cols)
matrix a;
int cols;
{
 matrix  mp = a;
 int  i, j;
 int  num_send = ringid,
      num_recv = num_send - 1;
 float  buf[rdim];

 if(ringid==0)
   num_recv = maxcell - 1;
```

```
  if(num_send)
   {
    /* send out cols of local columns */
    for(i=0; i<cols; i++)
      send_a_buf((char *) mp[cols - 1 -i], rdim*elsize);
   }

   if(num_recv && (ringid != 0))
    {
     for(i=0; i<num_recv; i++)
        for(j=0; j<cols; j++)
         {
          receive_a_buf((char *) buf, rdim*elsize);
          send_a_buf((char *) buf, rdim*elsize);
          }
    }

   if(ringid==0)
    {
      /* redistribute local columns in cell 0 */
      for(i=1; i<cols; i++)
        strcpy((char *) mp[i*maxcell], (char *) mp[i], rdim*elsize);
      for(i=1; i<=num_recv; i++)
        for(j=0; j<cols; j++)
          receive_a_buf((char *) mp[rdim - i -j*maxcell], rdim*elsize);
    } /* if */
} /* gather_g */


void free_mem(a, cn)
matrix  a;
int  cn;
{
  int  i;
  for(i=0; i<cn; i++)
    free(a[i]);
  free(a);
} /* free_mem */
```

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	April 1992	Final

4. TITLE AND SUBTITLE
AN IMPLEMENTATION OF THE MVDR BEAMFORMER ON THE INTEL iWARP SYSTEM

5. FUNDING NUMBERS
PE: 0602314N
WU: DN308291

6. AUTHOR(S)
J. Z. Lou

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Command, Control and Ocean Surveillance Center (NCCOSC)
RDT&E Division (NRaD)
San Diego, CA 92152–5000

8. PERFORMING ORGANIZATION REPORT NUMBER
NRaD TD 2282

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Office of Naval Technology
Office of the Chief of Naval Research
Arlington, VA 22217

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This report discusses the implementation of the minimum Variance Distortionless Response (MVDR) beamformer on the Intel iWarp system. A unidirectional ring and a full connection of processors were used in the implementations. The MVDR computation basically consists of a matrix factorization and a triangular system solver. A parallel Cholesky Factorization and a parallel QR factorization were implemented. A "fan-in" parallel algorithm was used for the triangle system solver. Using the unidirectional ring connection, the QR factorization achieved a 90% efficiency on 4 processors and a 64% efficiency on 16 processors: the Cholesky factorization achieved a 83% efficiency on 4 processors and a 44% efficiency on 16 processors. Using the unidirectonal ring connection and QR factorization, the MVDR beamformer achieved a 83% efficiency on 4 processors and 34% on 16 processors. Using the full connection of 4 processors, a 20% performance improvement has been obtained on MVDR as compared with the unidirectional ring connection. The parallel processing performance in terms of MFLOPS is still low, which is expected considering the low node performance on a C code using software release 2.3.

14. SUBJECT TERMS

15. NUMBER OF PAGES
40

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAME AS REPORT

21a. NAME OF RESPONSIBLE INDIVIDUAL	21b. TELEPHONE *(Include Area Code)*	21c. OFFICE SYMBOL
J. Z. Lou	(619) 553–2529	Code 421

INITIAL DISTRIBUTION

Code 0012	Patent Counsel	(1)
Code 0144	R. November	(1)
Code 144	V. Ware	(1)
Code 40	R. C. Kolb	(1)
Code 42	J. A. Salzmann	(1)
Code 421	D. L. Conwell	(1)
Code 421	J. Lou	(7)
Code 952B	J. Puleo	(1)
Code 961	Archive/Stock	(6)
Code 964B	Library	(2)

Defense Technical Information Center
Alexandria, VA 22304-6145 (4)

NCCOSC Washington Liaison Office
Washington, DC 20363-5100

Center for Naval Analyses
Alexandria, VA 22302-0268

Navy Acquisition, Research & Development
Information Center (NARDIC)
Washington, DC 20360-5000